Lecture 18: Time series

UBC 2022-23

Instructor: Mathias Lécuyer

Announcements

- HW8 released, due on April 12.

In [1]:
```
%load_ext autoreload
%autoreload 2
```

In [2]:
```
import sys
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.compose import ColumnTransformer, make_column_transformer
from sklearn.dummy import DummyClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import (
    TimeSeriesSplit,
    cross_val_score,
    cross_validate,
    train_test_split,
)
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, Standa

sys.path.append("../code/.")

plt.rcParams["font.size"] = 16
from datetime import datetime
```

Learning objectives

- Explain the pitfalls of train/test splitting with time series data.
- Appropriately split time series data, both train/test split and cross-validation.
- Perform time series feature engineering:
  - Encode time as various features in a tabular dataset
  - Create lag-based features
- Explain how can you forecast multiple time steps into the future.
- Explain the challenges of time series data with unequally spaced time points.

Motivation

- **Time series** is a collection of data points indexed in time order.
- Time series is everywhere:
  - Physical sciences (e.g., weather forecasting)
  - Economics, finance (e.g., stocks, market trends)
  - Engineering (e.g., energy consumption)
  - Social sciences
  - Sports analytics

Let's start with a simple example from Introduction to Machine Learning with Python book (https://learning.oreilly.com/library/view/introduction-to-machine/9781449369880/).

In New York city there is a network of bike rental stations with a subscription system. The stations are all around the city. The anonymized data is available here (https://ride.citibikenyc.com/system-data).

We will focus on the task of predicting how many people will rent a bicycle from a particular station for a given time and day. We might be interested in knowing this so that we know whether there will be any bikes left at the station for a particular day and time.

In [3]:
```
def load_citibike(file):
    data_mine = pd.read_csv(file)
    data_mine['one'] = 1
    data_mine['starttime'] = pd.to_datetime(data_mine.starttime)
    data_starttime = data_mine.set_index("starttime")
    data_resampled = data_starttime.resample("3h").sum(numeric_only=Tru

    return data_resampled.one

citibike = load_citibike("../data/201508-citibike-tripdata.csv")
citibike.head()
```

Out[3]:
```
starttime
2015-08-01 00:00:00    1135
2015-08-01 03:00:00     302
2015-08-01 06:00:00    1781
2015-08-01 09:00:00    7126
2015-08-01 12:00:00    8442
Freq: 3H, Name: one, dtype: int64
```

- The only feature we have is the date time feature.
  - Example: 2015-08-01 00:00:00
- The target is the number of rentals in the next 3 hours.
  - Example: 1135 rentals between 2015-08-01 00:00:00 and 2015-08-01 02:59:59

In [4]:
```
citibike.index.min()
```

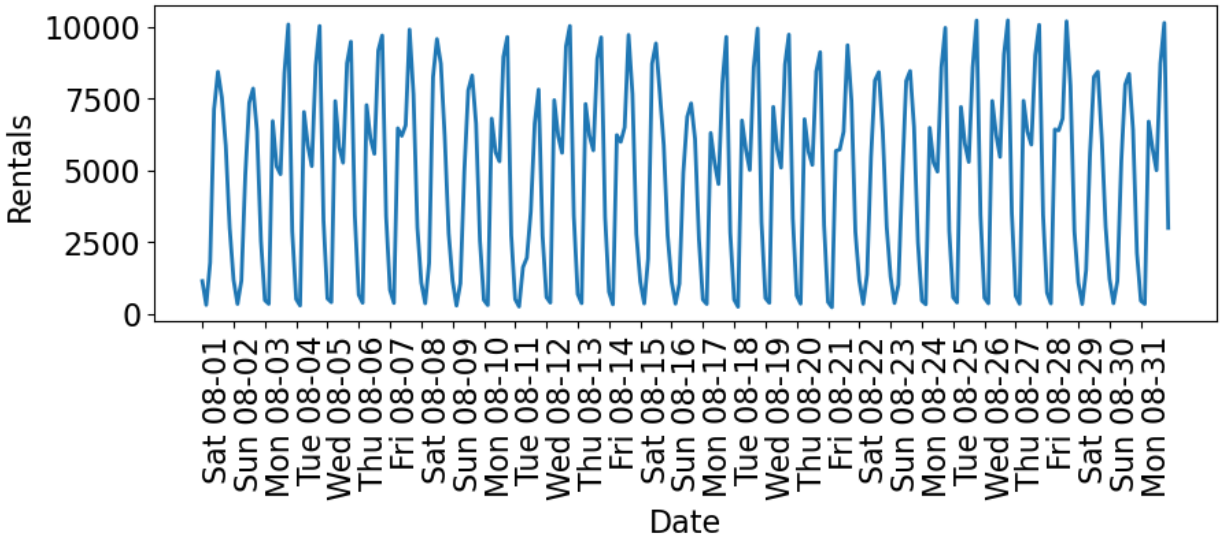
Out[4]:
```
Timestamp('2015-08-01 00:00:00', freq='3H')
```

In [5]:
```
citibike.index.max()
```

Out[5]:
```
Timestamp('2015-08-31 21:00:00', freq='3H')
```

We have data for August 2015.

In [6]:
```
plt.figure(figsize=(10, 3))
xticks = pd.date_range(start=citibike.index.min(), end=citibike.index.m
plt.xticks(xticks, xticks.strftime("%a %m-%d"), rotation=90, ha="left")
plt.plot(citibike, linewidth=2)
plt.xlabel("Date")
plt.ylabel("Rentals");
```



- We see the day and night pattern
- We see the weekend and weekday pattern

- Questions you might want to answer: How many people are likely to rent a bike at this station tomorrow at 3pm given everything we know about rentals in the past?
- We want to learn from the past and predict the future.

Train/test split for temporal data

- What will happen if we split this data the usual way?

In [7]:
```
train_df, test_df = train_test_split(citibike, test_size=0.2, random_st
```

In [8]:
```
1 test_df.head()
```

Out[8]:
```
starttime
2015-08-26 12:00:00    5469
2015-08-12 09:00:00    6199
2015-08-19 03:00:00     373
2015-08-07 12:00:00    6573
2015-08-03 09:00:00    5144
Name: one, dtype: int64
```

In [9]:
```
1 train_df.index.max()
```

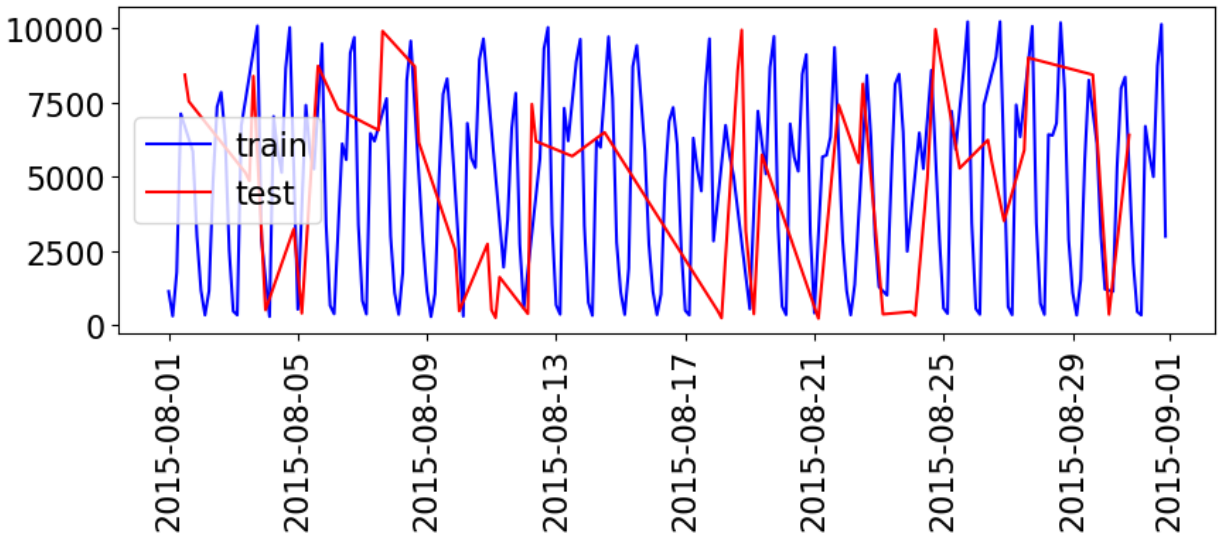
Out[9]:
```
Timestamp('2015-08-31 21:00:00')
```

In [10]:
```
1 test_df.index.min()
```

Out[10]:
```
Timestamp('2015-08-01 12:00:00')
```

- So, we are training on data that came after our test data!
- If we want to forecast, **we aren't allowed to know what happened in the future**!
- There may be cases where this is OK, e.g. if you aren't trying to forecast and just want to understand your data (maybe you're not even splitting).
- But, for our purposes, we want to avoid this.

In [11]:
```
1 plt.figure(figsize=(10, 3))
2 train_df_sort = train_df.sort_index()
3 test_df_sort = test_df.sort_index()
4
5 plt.plot(train_df_sort, "b", label="train")
6 plt.plot(test_df_sort, "r", label="test")
7 plt.xticks(rotation="vertical")
8 plt.legend();
```



We'll split the data as follows:

- We have total 248 data points.

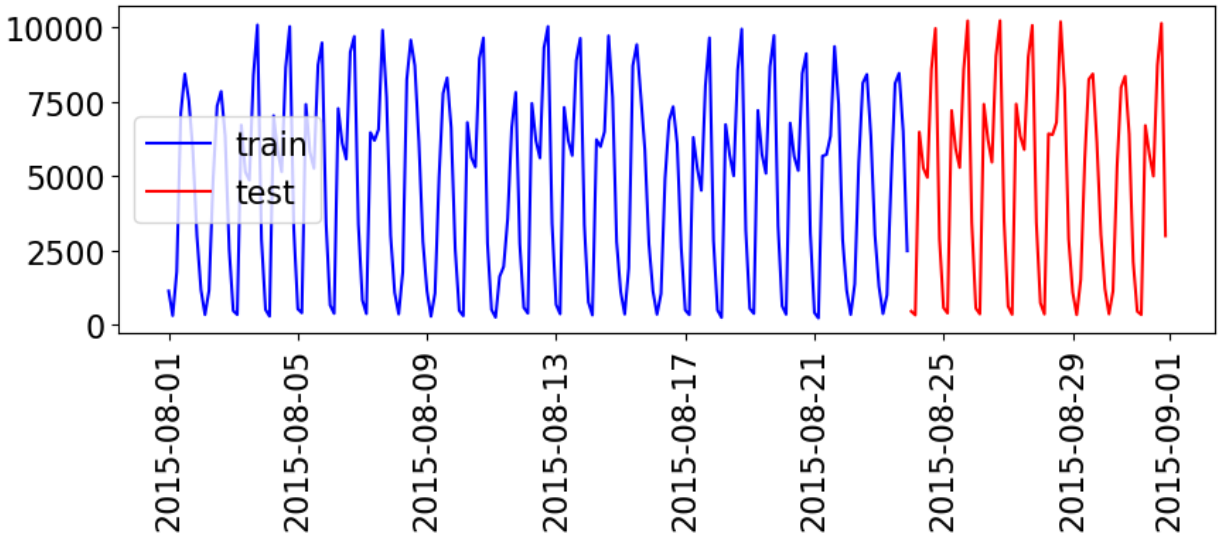
- We'll use the fist 184 data points corresponding to the first 23 days as training data - And the remaining 64 data points corresponding to the remaining 8 days as test data.

In [12]:
```
citibike.shape
```

Out[12]: (248,)

In [13]:
```
n_train = 184
train_df = citibike[:184]
test_df = citibike[184:]
```

In [14]:
```
plt.figure(figsize=(10, 3))
train_df_sort = train_df.sort_index()
test_df_sort = test_df.sort_index()

plt.plot(train_df_sort, "b", label="train")
plt.plot(test_df_sort, "r", label="test")
plt.xticks(rotation="vertical")
plt.legend();
```



- This split is looking reasonable now.

Training models

- In this toy data, we just have a single feature: the date time feature.
- We need to encode this feature if we want to build machine learning models.
- A common way that dates are stored on computers is using POSIX time, which is the number of seconds since January 1970 00:00:00 (this is beginning of Unix time).
- Let's start with encoding this feature as a single integer representing this POSIX time.

In [15]:
```
X = (
    citibike.index.astype("int64").values.reshape(-1, 1) // 10 ** 9
)  # convert to POSIX time by dividing by 10**9
y = citibike.values
```

In [16]:
```
y_train = train_df.values
y_test = test_df.values
# convert to POSIX time by dividing by 10**9
X_train = train_df.index.astype("int64").values.reshape(-1, 1) // 10 **
X_test = test_df.index.astype("int64").values.reshape(-1, 1) // 10 ** 9
```

In [17]:
```
X_train[:10]
```

Out[17]:
```
array([[1438387200],
       [1438398000],
       [1438408800],
       [1438419600],
       [1438430400],
       [1438441200],
       [1438452000],
       [1438462800],
       [1438473600],
       [1438484400]])
```

In [18]:
```
y_train[:10]
```

Out[18]:
```
array([1135,  302, 1781, 7126, 8442, 7544, 5858, 2991, 1173,  336])
```

- Our prediction task is a regression task.

Let's try random forest regression.

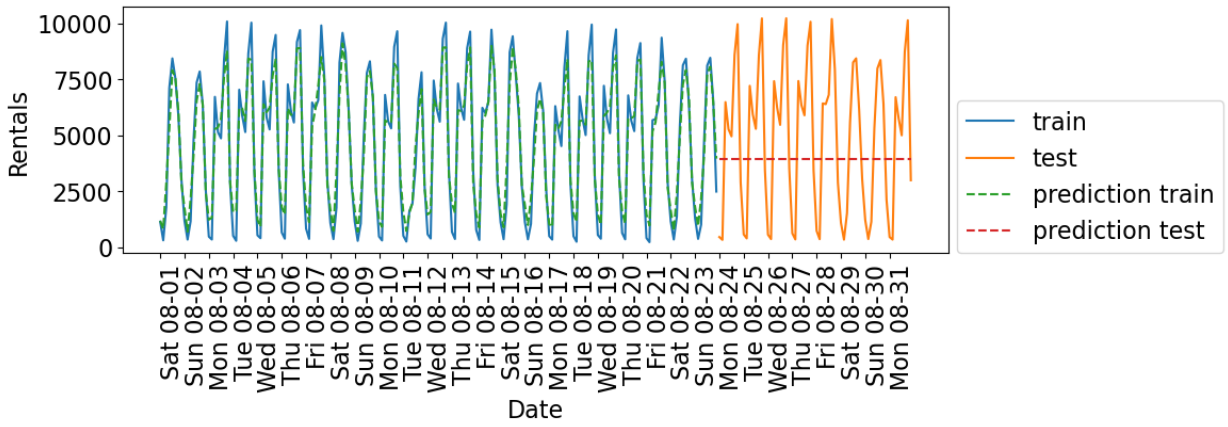
In [19]:
```
from sklearn.ensemble import RandomForestRegressor

regressor = RandomForestRegressor(n_estimators=100, random_state=0)
regressor.fit(X_train, y_train)

print("Train-set R^2: {:.2f}".format(regressor.score(X_train, y_train)))
print("Test-set R^2: {:.2f}".format(regressor.score(X_test, y_test)))
```

```
Train-set R^2: 0.94
Test-set R^2: -0.09
```

In [20]:
```
## Code credit: https://learning.oreilly.com/library/view/introduction-


def eval_on_features(features, target, regressor):
    # split the given features into a training and a test set
    X_train, X_test = features[:n_train], features[n_train:]
    # also split the target array
    y_train, y_test = target[:n_train], target[n_train:]
    regressor.fit(X_train, y_train)
    print("Train-set R^2: {:.2f}".format(regressor.score(X_train, y_tra
    print("Test-set R^2: {:.2f}".format(regressor.score(X_test, y_test)
    y_pred = regressor.predict(X_test)
    y_pred_train = regressor.predict(X_train)
    plt.figure(figsize=(10, 3))

    plt.xticks(range(0, len(X), 8), xticks.strftime("%a %m-%d"), rotati

    plt.plot(range(n_train), y_train, label="train")
    plt.plot(range(n_train, len(y_test) + n_train), y_test, "-", label=
    plt.plot(range(n_train), y_pred_train, "--", label="prediction trai

    plt.plot(
        range(n_train, len(y_test) + n_train), y_pred, "--", label="pre
    )
    plt.legend(loc=(1.01, 0))
    plt.xlabel("Date")
    plt.ylabel("Rentals")
```

In [21]:
```
from sklearn.ensemble import RandomForestRegressor

regressor = RandomForestRegressor(n_estimators=100, random_state=0)
eval_on_features(X, y, regressor)
```

```
Train-set R^2: 0.94
Test-set R^2: -0.09
```



- The predictions on the training set are pretty good
- But for the test data, a constant line is predicted ...
- What's going on?

- The model is based on only one feature: POSIX time feature.
- And the value of the POSIX time feature is outside the range of the feature values in the training set.
- Tree-based models cannot *extrapolate* to feature ranges outside the training data.
- The model predicted the target value of the closest point in the training set.

Can we come up with better features?

Feature engineering for date/time columns

- Note that our index is of this special type: `DateTimeIndex` (https://pandas.pydata.org/docs/reference/api/pandas.DatetimeIndex.html). We can extract all kinds of interesting information from it.

In [22]:
```
1 citibike.index
```

Out[22]:
```
DatetimeIndex(['2015-08-01 00:00:00', '2015-08-01 03:00:00',
               '2015-08-01 06:00:00', '2015-08-01 09:00:00',
               '2015-08-01 12:00:00', '2015-08-01 15:00:00',
               '2015-08-01 18:00:00', '2015-08-01 21:00:00',
               '2015-08-02 00:00:00', '2015-08-02 03:00:00',
               ...
               '2015-08-30 18:00:00', '2015-08-30 21:00:00',
               '2015-08-31 00:00:00', '2015-08-31 03:00:00',
               '2015-08-31 06:00:00', '2015-08-31 09:00:00',
               '2015-08-31 12:00:00', '2015-08-31 15:00:00',
               '2015-08-31 18:00:00', '2015-08-31 21:00:00'],
              dtype='datetime64[ns]', name='starttime', length=248, freq
='3H')
```

In [23]:
```
1 citibike.index.month_name()
```

Out[23]:
```
Index(['August', 'August', 'August', 'August', 'August', 'August', 'Augus
t',
       'August', 'August', 'August',
       ...
       'August', 'August', 'August', 'August', 'August', 'August', 'Augus
t',
       'August', 'August', 'August'],
      dtype='object', name='starttime', length=248)
```

In [24]:
```
1 citibike.index.dayofweek
```

Out[24]:
```
Int64Index([5, 5, 5, 5, 5, 5, 5, 5, 6, 6,
            ...
            6, 6, 0, 0, 0, 0, 0, 0, 0, 0],
           dtype='int64', name='starttime', length=248)
```

In [25]:
```
1 citibike.index.day_name()
```

Out[25]:
```
Index(['Saturday', 'Saturday', 'Saturday', 'Saturday', 'Saturday', 'Saturday',
       'Saturday', 'Saturday', 'Sunday', 'Sunday',
       ...
       'Sunday', 'Sunday', 'Monday', 'Monday', 'Monday', 'Monday', 'Monday',
       'Monday', 'Monday', 'Monday'],
      dtype='object', name='starttime', length=248)
```

In [26]:
```
1 citibike.index.hour
```

Out[26]:
```
Int64Index([ 0,  3,  6,  9, 12, 15, 18, 21,  0,  3,
            ...
            18, 21,  0,  3,  6,  9, 12, 15, 18, 21],
           dtype='int64', name='starttime', length=248)
```

- We noted before that the time of the day and day of the week seem quite important.
- Let's add these two features.

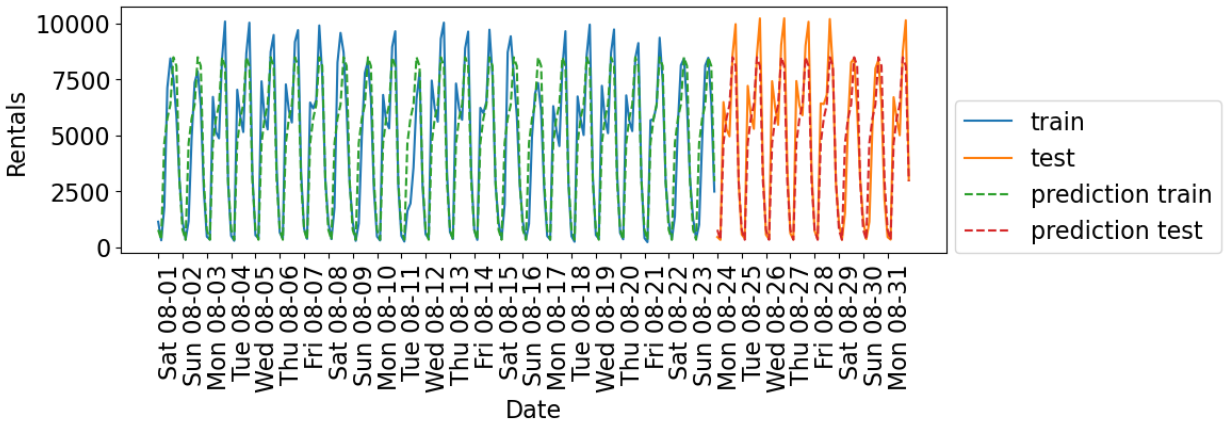
Let's first add the time of the day.

In [27]:
```
1 X_hour = citibike.index.hour.values.reshape(-1, 1)
2 X_hour[:10]
```

Out[27]:
```
array([[ 0],
       [ 3],
       [ 6],
       [ 9],
       [12],
       [15],
       [18],
       [21],
       [ 0],
       [ 3]])
```

In [28]:
```
1 eval_on_features(X_hour, y, regressor)
```

```
Train-set R^2: 0.82
Test-set R^2: 0.85
```

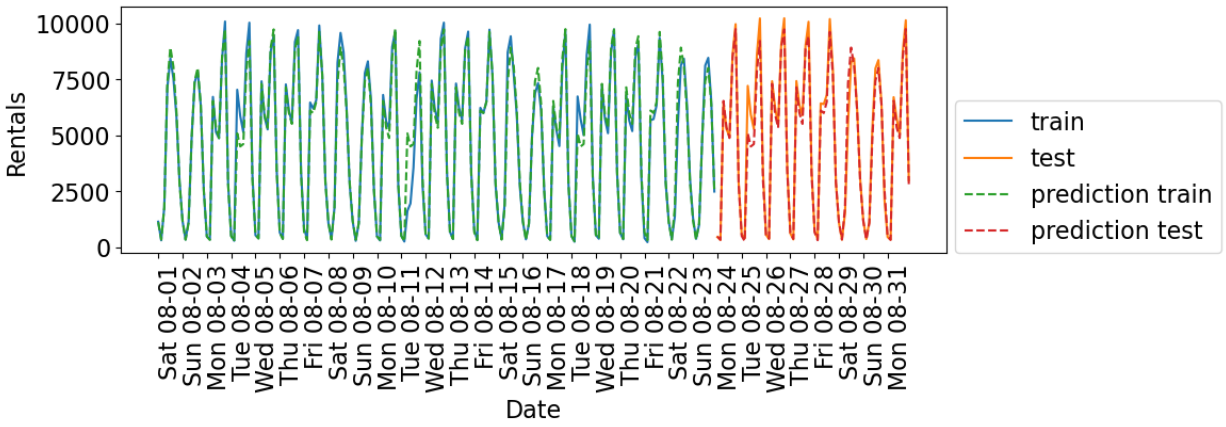


The scores are better than before.

Now let's add day of the week along with time of the day.

In [29]:
```
1 X_hour_week = np.hstack(
2     [
3         citibike.index.dayofweek.values.reshape(-1, 1),
4         citibike.index.hour.values.reshape(-1, 1),
5     ]
6 )
7 eval_on_features(X_hour_week, y, regressor)
```

```
Train-set R^2: 0.97
Test-set R^2: 0.98
```

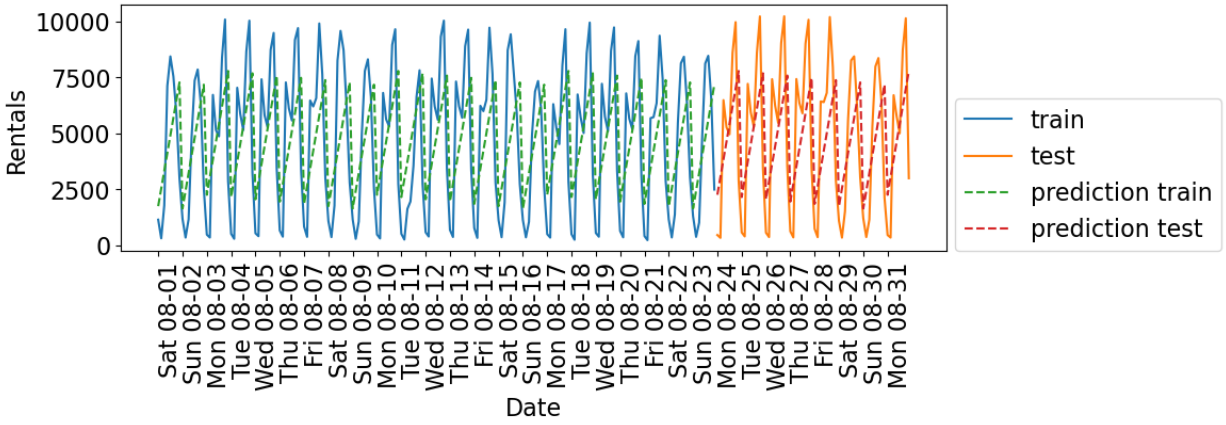


The results are much better. The time of the day and day of the week features are clearly helping.

- Do we need a complex model such as a random forest?
- Let's try `Ridge` with these features.

In [30]:
```
from sklearn.linear_model import Ridge

lr = Ridge()
eval_on_features(X_hour_week, y, lr)
```

```
Train-set R^2: 0.32
Test-set R^2: 0.33
```



- Ridge is performing poorly on the training as well as test data.
- It's not able to capture the periodic pattern.
- The reason is that we have encoded time of day using integers.
- A linear function can only learn a linear function of the time of day.
- What if we encode this feature as a categorical variable?

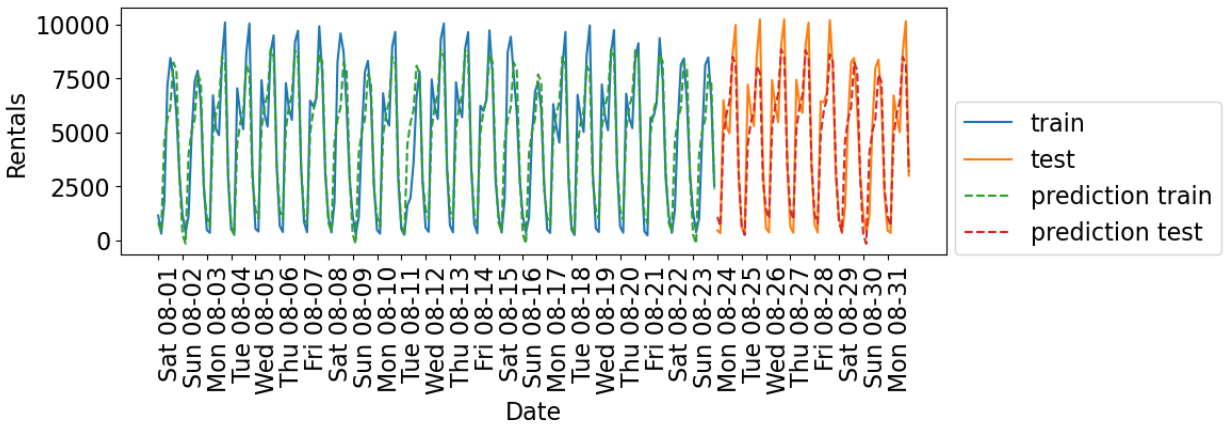
In [31]:
```
enc = OneHotEncoder()
X_hour_week_onehot = enc.fit_transform(X_hour_week).toarray()
```

In [32]:
```
X_hour_week_onehot
X_hour_week_onehot.shape
```

Out[32]:
```
(248, 15)
```

In [33]:
```
eval_on_features(X_hour_week_onehot, y, Ridge())
```

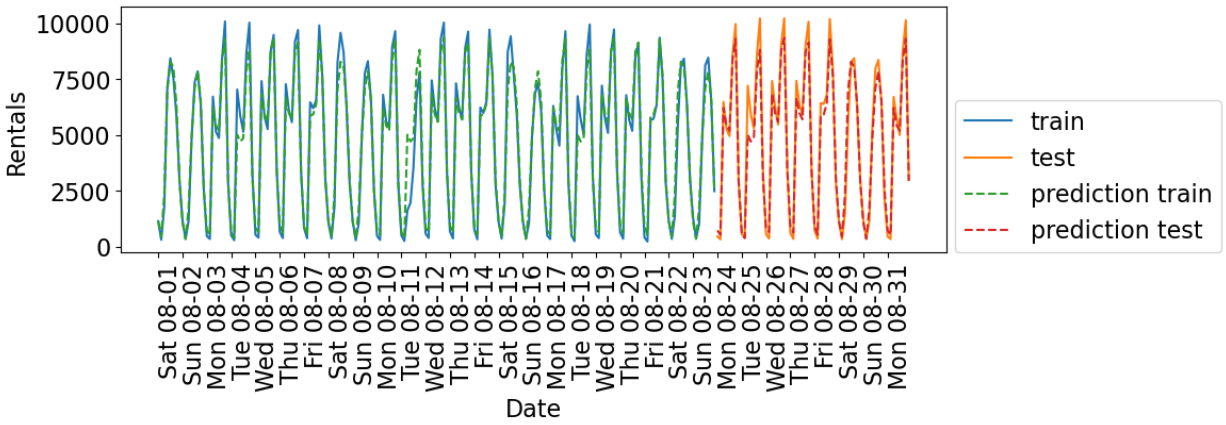
```
Train-set R^2: 0.84
Test-set R^2: 0.87
```



What if we add interaction features. We can do it using `sklearn`'s `PolynomialFeatures` transformer.

In [34]:
```
from sklearn.preprocessing import PolynomialFeatures

poly_transformer = PolynomialFeatures(
    degree=2, interaction_only=True, include_bias=False
)
X_hour_week_onehot_poly = poly_transformer.fit_transform(X_hour_week_on
lr = Ridge()
eval_on_features(X_hour_week_onehot_poly, y, lr)
```

```
Train-set R^2: 0.97
Test-set R^2: 0.97
```



In [35]:
```
X_hour_week_onehot_poly.shape
```

Out[35]:
```
(248, 120)
```

In [36]:
```
hour = ["%02d:00" % i for i in range(0, 24, 3)]
day = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
features = day + hour
features
```

Out[36]:
```
['Mon',
 'Tue',
 'Wed',
 'Thu',
 'Fri',
 'Sat',
 'Sun',
 '00:00',
 '03:00',
 '06:00',
 '09:00',
 '12:00',
 '15:00',
 '18:00',
 '21:00']
```

Let's examine the coefficients learned by `Ridge`.

In [37]:
```
features_poly = poly_transformer.get_feature_names_out(features)
features_nonzero = np.array(features_poly)[lr.coef_ != 0]
coef_nonzero = lr.coef_[lr.coef_ != 0]
```

In [38]:
```
coefs = pd.DataFrame(coef_nonzero, index=features_nonzero, columns=["Co
    "Coefficient", ascending=False
)
```

In [39]:
```
coefs[:10]
```

Out[39]:

	Coefficient
15:00	3228.129921
18:00	3012.775591
Sat 12:00	2449.171257
Sun 12:00	1753.500986
Sat 09:00	1568.393304
Wed 06:00	1527.908029
Mon 18:00	1473.445712
Thu 06:00	1380.854458
Tue 18:00	1355.142140
12:00	1251.901575

In [40]:
```
coefs[-10:]
```

Out[40]:

	Coefficient
Thu 00:00	-814.917196
Tue 12:00	-848.702348
Mon 12:00	-884.148776
Wed 00:00	-924.113625
Sat 18:00	-1228.127956
21:00	-1474.846457
Sat 06:00	-2455.263389
Sun 06:00	-2576.733659
00:00	-3353.192913
03:00	-3702.586614

- The coefficients make sense!
- If it's Saturday 09:00 or Wednesday 06:00, the model is likely to predict bigger number for rentals.

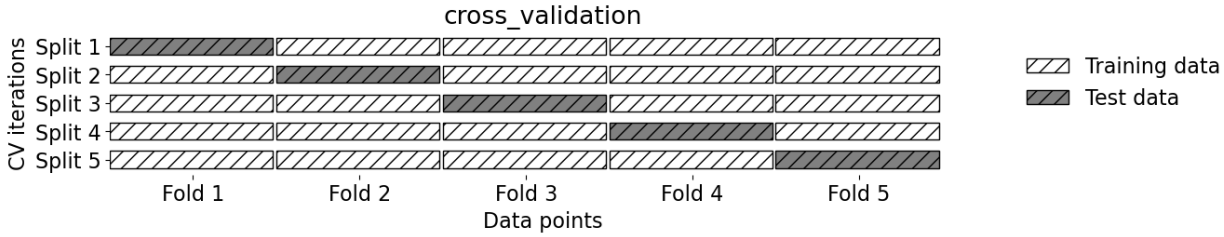
In [ ]:
```
1
```

Cross-validation

What about cross-validation?

- We can't do regular cross-validation if we don't want to be predicting the past.
- If you carry out regular cross-validation, you'll be predicting the past given future which is not a realistic scenario for the deployment data.

In [41]:
```
1 import mglearn_utils
2 mglearn_utils.plot_cross_validation()
```



There is `TimeSeriesSplit` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html) for time series data.

In [42]:
```
1 from sklearn.model_selection import TimeSeriesSplit
```

In [43]:
```
1 # Code from sklearn documentation
2 X_toy = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]])
3 y_toy = np.array([1, 2, 3, 4, 5, 6])
4 tscv = TimeSeriesSplit(n_splits=3)
5 for train, test in tscv.split(X_toy):
6     print("%s %s" % (train, test))
```

```
[0 1 2] [3]
[0 1 2 3] [4]
[0 1 2 3 4] [5]
```

Let's try it out with Ridge on the citibike data.

In [44]:
```
1 lr = Ridge()
```

In [45]:
```
scores = cross_validate(
    lr, X_hour_week_onehot_poly, y, cv=TimeSeriesSplit(n_splits=3), ret
)
pd.DataFrame(scores)
```

Out[45]:

	fit_time	score_time	test_score	train_score
0	0.000553	0.000210	0.807857	0.947222
1	0.000605	0.000139	0.943975	0.946537
2	0.000650	0.000124	0.956777	0.961063
3	0.000667	0.000115	0.953726	0.965518
4	0.000781	0.000115	0.978614	0.967371

A more complicated dataset (5 min)

Rain in Australia (https://www.kaggle.com/jsphyg/weather-dataset-rattle-package) dataset.
Predicting whether or not it will rain tomorrow based on today's measurements.

In [46]:
```
rain_df = pd.read_csv("../data/weatherAUS.csv")
rain_df.head()
```

Out[46]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpe
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	4
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	4
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	4
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	2
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	4

5 rows × 23 columns

In [47]:
```
rain_df.shape
```

Out[47]:
```
(145460, 23)
```

Goals

- Can the date/time features help us predict the target value?
- Can we **forecast** into the future?

In [48]:
```
1 rain_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145460 entries, 0 to 145459
Data columns (total 23 columns):
 #   Column         Non-Null Count   Dtype  
---  ------         --------------   -----  
 0   Date           145460 non-null  object 
 1   Location       145460 non-null  object 
 2   MinTemp        143975 non-null  float64
 3   MaxTemp        144199 non-null  float64
 4   Rainfall       142199 non-null  float64
 5   Evaporation    82670 non-null   float64
 6   Sunshine       75625 non-null   float64
 7   WindGustDir    135134 non-null  object 
 8   WindGustSpeed  135197 non-null  float64
 9   WindDir9am     134894 non-null  object 
 10  WindDir3pm     141232 non-null  object 
 11  WindSpeed9am   143693 non-null  float64
 12  WindSpeed3pm   142398 non-null  float64
 13  Humidity9am    142806 non-null  float64
 14  Humidity3pm    140953 non-null  float64
 15  Pressure9am    130395 non-null  float64
 16  Pressure3pm    130432 non-null  float64
 17  Cloud9am       89572 non-null   float64
 18  Cloud3pm       86102 non-null   float64
 19  Temp9am        143693 non-null  float64
 20  Temp3pm        141851 non-null  float64
 21  RainToday      142199 non-null  object 
 22  RainTomorrow   142193 non-null  object 
dtypes: float64(16), object(7)
memory usage: 25.5+ MB
```

In [49]:
```
rain_df.describe(include="all")
```

Out[49]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine
count	145460	145460	143975.000000	144199.000000	142199.000000	82670.000000	75625.000000
unique	3436	49	NaN	NaN	NaN	NaN	NaN
top	2013-11-12	Canberra	NaN	NaN	NaN	NaN	NaN
freq	49	3436	NaN	NaN	NaN	NaN	NaN
mean	NaN	NaN	12.194034	23.221348	2.360918	5.468232	7.611178
std	NaN	NaN	6.398495	7.119049	8.478060	4.193704	3.785483
min	NaN	NaN	-8.500000	-4.800000	0.000000	0.000000	0.000000
25%	NaN	NaN	7.600000	17.900000	0.000000	2.600000	4.800000
50%	NaN	NaN	12.000000	22.600000	0.000000	4.800000	8.400000
75%	NaN	NaN	16.900000	28.200000	0.800000	7.400000	10.600000
max	NaN	NaN	33.900000	48.100000	371.000000	145.000000	14.500000

11 rows × 23 columns

- A number of missing values.
- Some target values are also missing. I'm dropping these rows.

In [50]:
```
rain_df = rain_df[rain_df["RainTomorrow"].notna()]
rain_df.shape
```

Out[50]:
```
(142193, 23)
```

Parsing datetimes

- In general, datetimes are a huge pain! Think of all the formats: MM-DD-YY, DD-MM-YY, YY-MM-DD, MM/DD/YY, DD/MM/YY, DD/MM/YYYY, 20:45, 8:45am, 8:45 PM, 8:45a, 08:00, 8:10:20, ...
- No, seriously, dealing with datetimes is THE WORST.
  - Time zones.
  - Daylight savings...
- Thankfully, pandas does a pretty good job here.

In [51]:
```
dates_rain = pd.to_datetime(rain_df["Date"])
dates_rain
```

Out[51]:
```
0         2008-12-01
1         2008-12-02
2         2008-12-03
3         2008-12-04
4         2008-12-05
             ...
145454    2017-06-20
145455    2017-06-21
145456    2017-06-22
145457    2017-06-23
145458    2017-06-24
Name: Date, Length: 142193, dtype: datetime64[ns]
```

They are all the same format, so we can also compare dates:

In [52]:
```
dates_rain[1] - dates_rain[0]
```

Out[52]:
```
Timedelta('1 days 00:00:00')
```

In [53]:
```
dates_rain[1] > dates_rain[0]
```

Out[53]:
```
True
```

In [54]:
```
(dates_rain[1] - dates_rain[0]).total_seconds()
```

Out[54]:
```
86400.0
```

We can also easily extract information from the date columns.

In [55]:
```
dates_rain[1]
```

Out[55]:
```
Timestamp('2008-12-02 00:00:00')
```

In [56]:
```
dates_rain[1].month_name()
```

Out[56]:
```
'December'
```

In [57]:
```
dates_rain[1].day_name()
```

Out[57]:
```
'Tuesday'
```

In [58]:
```
dates_rain[1].is_year_end
```

Out[58]:
```
False
```

In [59]:
```
dates_rain[1].is_leap_year
```

Out[59]:
```
True
```

Above pandas identified the date column automatically. You can tell pandas to parse the dates when reading in the CSV:

In [60]:
```
rain_df = pd.read_csv("../data/weatherAUS.csv", parse_dates=["Date"])
rain_df.head()
```

Out[60]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpe
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	4
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	4
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	4
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	2
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	4

5 rows × 23 columns

In [61]:
```
rain_df = rain_df[rain_df["RainTomorrow"].notna()]
rain_df.shape
```

Out[61]:
```
(142193, 23)
```

In [62]:
```
rain_df["Date"].head()
```

Out[62]:
```
0   2008-12-01
1   2008-12-02
2   2008-12-03
3   2008-12-04
4   2008-12-05
Name: Date, dtype: datetime64[ns]
```

Train/test splits

- Remember that we should not be calling the usual `train_test_split` with shuffling because
- If we want to forecast, we aren't allowed to know what happened in the future!

In [63]:
```
rain_df["Date"].min()
```

Out[63]:
```
Timestamp('2007-11-01 00:00:00')
```

In [64]:
```
rain_df["Date"].max()
```

Out[64]:
```
Timestamp('2017-06-25 00:00:00')
```

- It looks like we have 10 years of data.

- Let's use the last 2 years for test.

In [65]:
```
train_df = rain_df.query("Date <= 20150630")
test_df = rain_df.query("Date >  20150630")
```

In [66]:
```
len(train_df)
```

Out[66]: 107502

In [67]:
```
len(test_df)
```

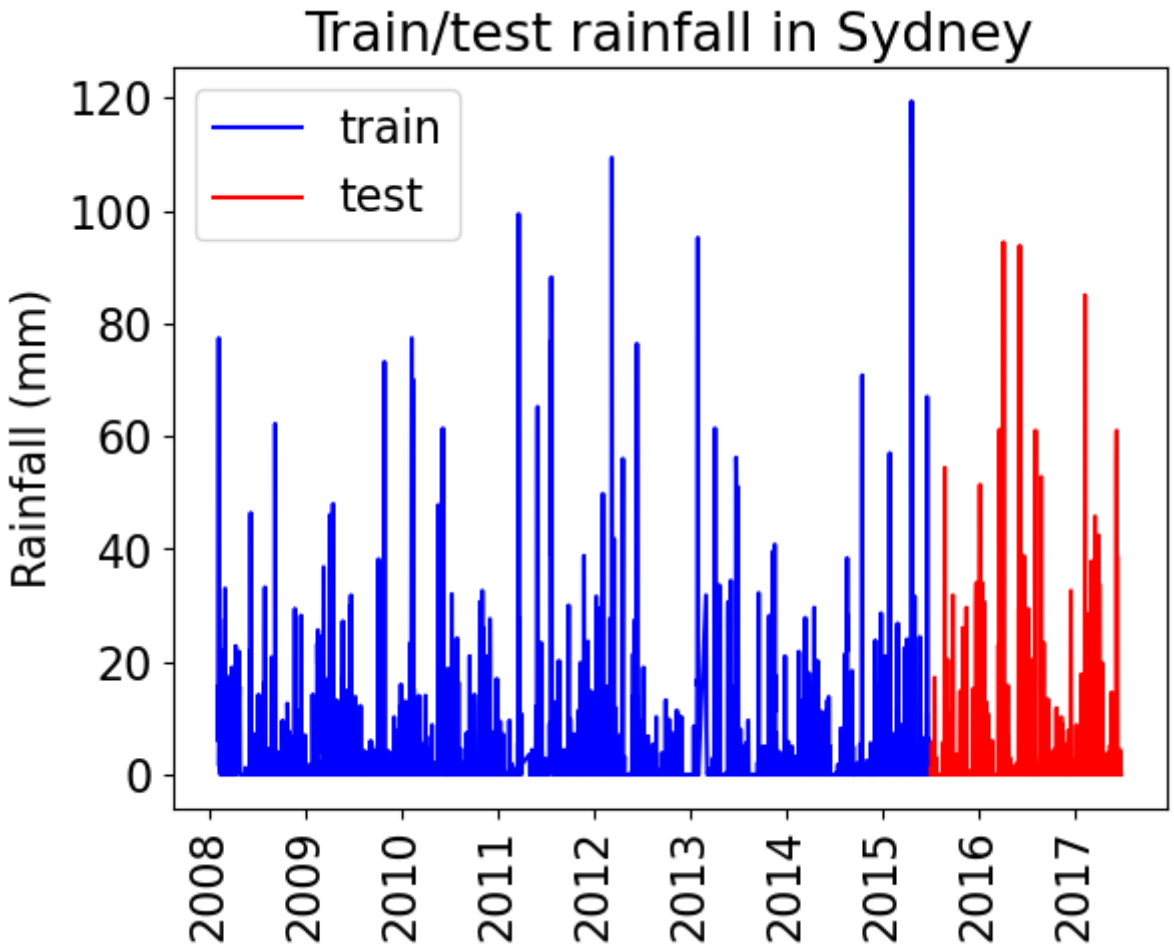
Out[67]: 34691

In [68]:
```
len(test_df) / (len(train_df) + len(test_df))
```

Out[68]: 0.24397122221206388

As we can see, we're still using about 25% of our data as test data.

In [69]:
```
train_df_sort = train_df.query("Location == 'Sydney'").sort_values(by="
test_df_sort = test_df.query("Location == 'Sydney'").sort_values(by="Da

plt.plot(train_df_sort["Date"], train_df_sort["Rainfall"], "b", label="
plt.plot(test_df_sort["Date"], test_df_sort["Rainfall"], "r", label="te
plt.xticks(rotation="vertical")
plt.legend()
plt.ylabel("Rainfall (mm)")
plt.title("Train/test rainfall in Sydney");
```



We're learning relationships from the blue part; predicting only using features in the red part from the day before.

Let's define a preprocessor with a column transformer.

In [70]:
```
train_df.columns
```

Out[70]:
```
Index(['Date', 'Location', 'MinTemp', 'MaxTemp', 'Rainfall', 'Evaporatio
n',
       'Sunshine', 'WindGustDir', 'WindGustSpeed', 'WindDir9am', 'WindDir
3pm',
       'WindSpeed9am', 'WindSpeed3pm', 'Humidity9am', 'Humidity3pm',
       'Pressure9am', 'Pressure3pm', 'Cloud9am', 'Cloud3pm', 'Temp9am',
       'Temp3pm', 'RainToday', 'RainTomorrow'],
      dtype='object')
```

In [71]:
```
1 train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 107502 entries, 0 to 144733
Data columns (total 23 columns):
 #   Column         Non-Null Count   Dtype
---  ------         --------------   -----
 0   Date           107502 non-null  datetime64[ns]
 1   Location       107502 non-null  object
 2   MinTemp        107050 non-null  float64
 3   MaxTemp        107292 non-null  float64
 4   Rainfall       106424 non-null  float64
 5   Evaporation    66221 non-null   float64
 6   Sunshine       62320 non-null   float64
 7   WindGustDir    100103 non-null  object
 8   WindGustSpeed  100146 non-null  float64
 9   WindDir9am     99515 non-null   object
 10  WindDir3pm     105314 non-null  object
 11  WindSpeed9am   106322 non-null  float64
 12  WindSpeed3pm   106319 non-null  float64
 13  Humidity9am    106112 non-null  float64
 14  Humidity3pm    106180 non-null  float64
 15  Pressure9am    97217 non-null   float64
 16  Pressure3pm    97253 non-null   float64
 17  Cloud9am       68523 non-null   float64
 18  Cloud3pm       67501 non-null   float64
 19  Temp9am        106705 non-null  float64
 20  Temp3pm        106816 non-null  float64
 21  RainToday      106424 non-null  object
 22  RainTomorrow   107502 non-null  object
dtypes: datetime64[ns](1), float64(16), object(6)
memory usage: 19.7+ MB
```

- We have missing data.
- We have categorical features and numeric features.

- Let's define feature types.
- Let's start with dropping the date column and treating it as a usual supervised machine learning problem.

In [72]:

```
numeric_features = [
    "MinTemp",
    "MaxTemp",
    "Rainfall",
    "Evaporation",
    "Sunshine",
    "WindGustSpeed",
    "WindSpeed9am",
    "WindSpeed3pm",
    "Humidity9am",
    "Humidity3pm",
    "Pressure9am",
    "Pressure3pm",
    "Cloud9am",
    "Cloud3pm",
    "Temp9am",
    "Temp3pm",
]
categorical_features = [
    "Location",
    "WindGustDir",
    "WindDir9am",
    "WindDir3pm",
    "RainToday",
]
drop_features = [
    "Date",
    "RainTomorrow",
]
```

In [73]:

```
def preprocess_features(
    train_df,
    test_df,
    numeric_features,
    categorical_features,
    drop_features,
):

    all_features = set(numeric_features + categorical_features + drop_f
    if set(train_df.columns) != all_features:
        print("Missing columns", set(train_df.columns) - all_features)
        print("Extra columns", all_features - set(train_df.columns))
        raise Exception("Columns do not match")

    numeric_transformer = make_pipeline(
        SimpleImputer(strategy="median"), StandardScaler()
    )
    categorical_transformer = make_pipeline(
        SimpleImputer(strategy="constant", fill_value="?"),
        OneHotEncoder(handle_unknown="ignore", sparse=False),
    )

    preprocessor = make_column_transformer(
        (numeric_transformer, numeric_features),
        (categorical_transformer, categorical_features),
        ("drop", drop_features),
    )
    preprocessor.fit(train_df)
    ohe_feature_names = (
        preprocessor.named_transformers_["pipeline-2"]
        .named_steps["onehotencoder"]
        .get_feature_names_out()
        .tolist()
    )
    new_columns = numeric_features + ohe_feature_names

    X_train_enc = pd.DataFrame(
        preprocessor.transform(train_df), index=train_df.index, columns
    )
    X_test_enc = pd.DataFrame(
        preprocessor.transform(test_df), index=test_df.index, columns=n
    )

    y_train = train_df["RainTomorrow"]
    y_test = test_df["RainTomorrow"]

    return X_train_enc, y_train, X_test_enc, y_test, preprocessor
```

In [74]:
```
X_train_enc, y_train, X_test_enc, y_test, preprocessor = preprocess_fea
    train_df,
    test_df,
    numeric_features,
    categorical_features,
    drop_features,
)
```

In [75]:
```
X_train_enc.head()
```

Out[75]:

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustSpeed	WindSpeed9am	WindSp
0	0.204302	-0.027112	-0.205323	-0.140641	0.160729	0.298612	0.666166	0
1	-0.741037	0.287031	-0.275008	-0.140641	0.160729	0.298612	-1.125617	0
2	0.125523	0.372706	-0.275008	-0.140641	0.160729	0.450132	0.554180	0
3	-0.457435	0.701128	-0.275008	-0.140641	0.160729	-1.216596	-0.341712	-1
4	0.850283	1.315134	-0.158867	-0.140641	0.160729	0.071330	-0.789657	0

5 rows × 119 columns

Baseline

`DummyClassifier`

In [76]:
```
dc = DummyClassifier(strategy="prior")
dc.fit(train_df, y_train);
```

In [77]:
```
dc.score(train_df, y_train)
```

Out[77]:
```
0.7750553478074826
```

In [78]:
```
y_train.value_counts()
```

Out[78]:
```
No     83320
Yes    24182
Name: RainTomorrow, dtype: int64
```

In [79]:
```
dc.score(test_df, y_test)
```

Out[79]:
```
0.7781845435415525
```

LogisticRegression

The function below trains a logistic regression model on the train set, reports train and test scores, and returns learned coefficients as a dataframe.

In [80]:
```
def score_lr_print_coeff(preprocessor, train_df, y_train, test_df, y_te
    lr_pipe = make_pipeline(preprocessor, LogisticRegression(max_iter=1
    lr_pipe.fit(train_df, y_train)
    print("Train score: {:.2f}".format(lr_pipe.score(train_df, y_train)
    print("Test score: {:.2f}".format(lr_pipe.score(test_df, y_test)))
    lr_coef = pd.DataFrame(
        data=lr_pipe.named_steps["logisticregression"].coef_.flatten(),
        index=X_train_enc.columns,
        columns=["Coef"],
    )
    return lr_coef.sort_values(by="Coef", ascending=False)
```

In [81]:
```
score_lr_print_coeff(preprocessor, train_df, y_train, test_df, y_test,
```

```
Train score: 0.85
Test score: 0.84
```

Out[81]:

	Coef
Humidity3pm	1.243231
x4_?	0.924657
Pressure9am	0.865428
x0_Witchcliffe	0.729015
WindGustSpeed	0.720465
...	...
x0_Townsville	-0.718734
x0_Katherine	-0.726151
x0_Wollongong	-0.748688
x0_MountGinini	-0.964870
Pressure3pm	-1.221746

119 rows × 1 columns

Cross-validation

- We can carry out cross-validation using `TimeSeriesSplit` (https://scikit-learn.org/stable/modules/cross_validation.html#time-series-split).
- However, things are actually more complicated here because this dataset has **multiple time series**, one per location.

In [82]:
```
1 train_df
```

Out[82]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGu
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	
...	...	...	...	...	...	...	...	...	
144729	2015-06-26	Uluru	3.8	18.3	0.0	NaN	NaN	E	
144730	2015-06-27	Uluru	2.5	17.1	0.0	NaN	NaN	E	
144731	2015-06-28	Uluru	4.5	19.6	0.0	NaN	NaN	ENE	
144732	2015-06-29	Uluru	7.6	22.0	0.0	NaN	NaN	ESE	
144733	2015-06-30	Uluru	6.8	21.1	0.0	NaN	NaN	ESE	

107502 rows × 23 columns

In [83]:
```
1 train_df.sort_values(by=["Date", "Location"]).head()
```

Out[83]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGus
45587	2007-11-01	Canberra	8.0	24.3	0.0	3.4	6.3	NW	
45588	2007-11-02	Canberra	14.0	26.9	3.6	4.4	9.7	ENE	
45589	2007-11-03	Canberra	13.7	23.4	3.6	5.8	3.3	NW	
45590	2007-11-04	Canberra	13.3	15.5	39.8	7.2	9.1	NW	
45591	2007-11-05	Canberra	7.6	16.1	2.8	5.6	10.6	SSE	

5 rows × 23 columns

In [84]:
```
train_df.sort_values(by=["Date"])
```

Out[84]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir
45587	2007-11-01	Canberra	8.0	24.3	0.0	3.4	6.3	NW
45588	2007-11-02	Canberra	14.0	26.9	3.6	4.4	9.7	ENE
45589	2007-11-03	Canberra	13.7	23.4	3.6	5.8	3.3	NW
45590	2007-11-04	Canberra	13.3	15.5	39.8	7.2	9.1	NW
45591	2007-11-05	Canberra	7.6	16.1	2.8	5.6	10.6	SSE
...	...	...	...	...	...	...	...	...
57415	2015-06-30	Ballarat	-0.3	10.5	0.0	NaN	NaN	S
119911	2015-06-30	PerthAirport	10.1	23.5	0.0	3.2	5.8	NNE
60455	2015-06-30	Bendigo	0.3	11.4	0.0	NaN	NaN	W
66473	2015-06-30	MelbourneAirport	3.2	13.2	0.0	0.8	3.9	N
144733	2015-06-30	Uluru	6.8	21.1	0.0	NaN	NaN	ESE

107502 rows × 23 columns

- It seems the dataframe is sorted by location, and then time.
- Our approach today will be to ignore the fact that we have multiple time series and just OHE the location
- We'll have multiple measurements for a given timestamp, and that's OK.
- But, `TimeSeriesSplit` expects the dataframe to be sorted by date so...

In [85]:
```
train_df_ordered = train_df.sort_values(by=["Date"])
y_train_ordered = train_df_ordered["RainTomorrow"]
```

In [86]:
```
1 train_df_ordered
```

Out[86]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir
45587	2007-11-01	Canberra	8.0	24.3	0.0	3.4	6.3	NW
45588	2007-11-02	Canberra	14.0	26.9	3.6	4.4	9.7	ENE
45589	2007-11-03	Canberra	13.7	23.4	3.6	5.8	3.3	NW
45590	2007-11-04	Canberra	13.3	15.5	39.8	7.2	9.1	NW
45591	2007-11-05	Canberra	7.6	16.1	2.8	5.6	10.6	SSE
...	...	...	...	...	...	...	...	...
57415	2015-06-30	Ballarat	-0.3	10.5	0.0	NaN	NaN	S
119911	2015-06-30	PerthAirport	10.1	23.5	0.0	3.2	5.8	NNE
60455	2015-06-30	Bendigo	0.3	11.4	0.0	NaN	NaN	W
66473	2015-06-30	MelbourneAirport	3.2	13.2	0.0	0.8	3.9	N
144733	2015-06-30	Uluru	6.8	21.1	0.0	NaN	NaN	ESE

107502 rows × 23 columns

In [87]:
```
1 lr_pipe = make_pipeline(preprocessor, LogisticRegression(max_iter=1000)
2 cross_val_score(lr_pipe, train_df_ordered, y_train_ordered, cv=TimeSeri
```

Out[87]:
```
0.8478874811631412
```

In [ ]:
```
1
```

Encoding date/time as feature(s)

- Can we use the `Date` to help us predict the target?
- Probably! E.g. different amounts of rain in different seasons.
- This is feature engineering!

Encoding time as an number

- Idea 1: create a column of "days since Nov 1, 2007".

In [88]:
```
train_df = rain_df.query("Date <= 20150630")
test_df = rain_df.query("Date >  20150630")
```

In [89]:
```
first_day = train_df["Date"].min()

train_df = train_df.assign(
    Days_since=train_df["Date"].apply(lambda x: (x - first_day).days)
)
test_df = test_df.assign(
    Days_since=test_df["Date"].apply(lambda x: (x - first_day).days)
)
```

In [90]:
```
train_df.sort_values(by="Date").head()
```

Out[90]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGus
45587	2007-11-01	Canberra	8.0	24.3	0.0	3.4	6.3	NW	
45588	2007-11-02	Canberra	14.0	26.9	3.6	4.4	9.7	ENE	
45589	2007-11-03	Canberra	13.7	23.4	3.6	5.8	3.3	NW	
45590	2007-11-04	Canberra	13.3	15.5	39.8	7.2	9.1	NW	
45591	2007-11-05	Canberra	7.6	16.1	2.8	5.6	10.6	SSE	

5 rows × 24 columns

In [91]:
```
X_train_enc, y_train, X_test_enc, y_test, preprocessor = preprocess_fea
    train_df,
    test_df,
    numeric_features + ["Days_since"],
    categorical_features,
    drop_features,
)
```

In [92]:
```
1 score_lr_print_coeff(preprocessor, train_df, y_train, test_df, y_test,
```

```
Train score: 0.85
Test score: 0.84
```

Out[92]:

	Coef
Humidity3pm	1.243081
x4_?	0.931186
Pressure9am	0.864258
x0_Witchcliffe	0.730547
WindGustSpeed	0.720084
...	...
x0_Townsville	-0.716479
x0_Katherine	-0.737178
x0_Wollongong	-0.746131
x0_MountGinini	-0.963908
Pressure3pm	-1.221575

120 rows × 1 columns

- Not much improvement in the scores
- Can you think of other ways to generate features from the `Date` column?
- What about the month - that seems relevant. How should we encode the month?

Another idea month: encode as a categorical variable?

One-hot encoding of the month

In [93]:
```
1 train_df = rain_df.query("Date <= 20150630")
2 test_df = rain_df.query("Date >  20150630")
```

In [94]:
```
1 train_df = train_df.assign(
2     Month=train_df["Date"].apply(lambda x: x.month_name())
3 )  # x.month_name() to get the actual string
4 test_df = test_df.assign(Month=test_df["Date"].apply(lambda x: x.month_
```

In [95]:
```
train_df[["Date", "Month"]].sort_values(by="Month")
```

Out[95]:

	Date	Month
62657	2013-04-14	April
115089	2010-04-15	April
115090	2010-04-16	April
115091	2010-04-17	April
115092	2010-04-18	April
...	...	...
128828	2014-09-20	September
128829	2014-09-21	September
128830	2014-09-22	September
128820	2014-09-12	September
72036	2013-09-29	September

107502 rows × 2 columns

In [96]:
```
X_train_enc, y_train, X_test_enc, y_test, preprocessor = preprocess_fea
    train_df, test_df, numeric_features, categorical_features + ["Month
)
```

In [97]:
```
score_lr_print_coeff(preprocessor, train_df, y_train, test_df, y_test,
```

```
Train score: 0.85
Test score: 0.84
```

Out[97]:

	Coef
Humidity3pm	1.266933
x4_?	0.944802
Pressure9am	0.799138
x0_Witchcliffe	0.749015
WindGustSpeed	0.705677
...	...
x0_Darwin	-0.735810
x0_Wollongong	-0.748377
x0_Townsville	-0.903095
x0_Katherine	-0.929008
Pressure3pm	-1.182011

131 rows × 1 columns

One-hot encoding seasons

How about just summer/winter as a feature?

In [98]:
```
def get_season(month):
    WINTER_MONTHS = ["June", "July", "August"]
    AUTUMN_MONTHS = ["March", "April", "May"]
    SUMMER_MONTHS = ["December", "January", "February"]
    SPRING_MONTHS = ["September", "October", "November"]
    if month in WINTER_MONTHS:
        return "Winter"
    elif month in AUTUMN_MONTHS:
        return "Autumn"
    elif month in SUMMER_MONTHS:
        return "Summer"
    else:
        return "Fall"
```

In [99]:
```
train_df = train_df.assign(Season=train_df["Month"].apply(get_season))
test_df = test_df.assign(Season=test_df["Month"].apply(get_season))
```

In [100]:
```
train_df
```

Out[100]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGu
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	
...	...	...	...	...	...	...	...	...	
144729	2015-06-26	Uluru	3.8	18.3	0.0	NaN	NaN	E	
144730	2015-06-27	Uluru	2.5	17.1	0.0	NaN	NaN	E	
144731	2015-06-28	Uluru	4.5	19.6	0.0	NaN	NaN	ENE	
144732	2015-06-29	Uluru	7.6	22.0	0.0	NaN	NaN	ESE	
144733	2015-06-30	Uluru	6.8	21.1	0.0	NaN	NaN	ESE	

107502 rows × 25 columns

In [101]:
```
X_train_enc, y_train, X_test_enc, y_test, preprocessor = preprocess_fea
    train_df,
    test_df,
    numeric_features,
    categorical_features + ["Season"],
    drop_features + ["Month"],
)
```

In [102]:
```
X_train_enc.columns
```

Out[102]:
```
Index(['MinTemp', 'MaxTemp', 'Rainfall', 'Evaporation', 'Sunshine',
       'WindGustSpeed', 'WindSpeed9am', 'WindSpeed3pm', 'Humidity9am',
       'Humidity3pm',
       ...
       'x3_W', 'x3_WNW', 'x3_WSW', 'x4_?', 'x4_No', 'x4_Yes', 'x5_Autum
n',
       'x5_Fall', 'x5_Summer', 'x5_Winter'],
      dtype='object', length=123)
```

In [103]:
```
coeff_df = score_lr_print_coeff(
    preprocessor, train_df, y_train, test_df, y_test, X_train_enc
)
```

```
Train score: 0.85
Test score: 0.84
```

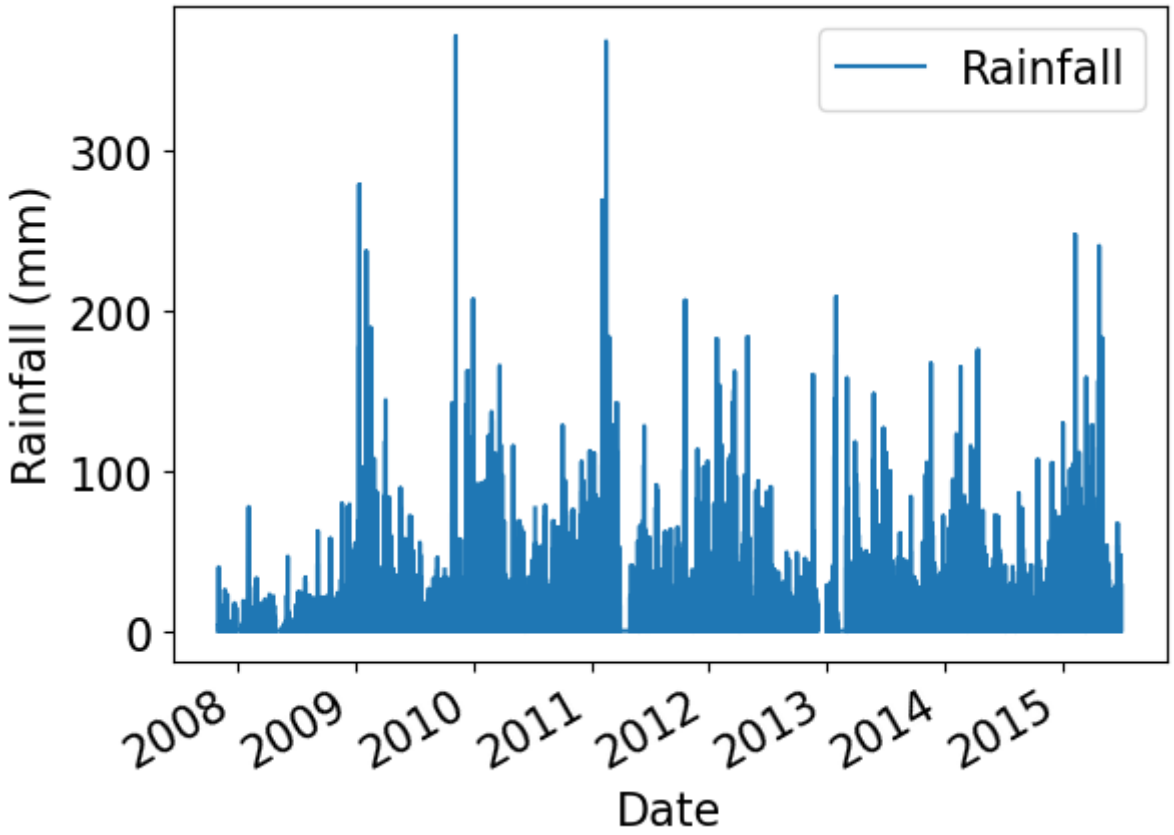
In [104]:
```
coeff_df.loc[["x5_Fall", "x5_Summer", "x5_Winter", "x5_Autumn"]]
```

Out[104]:

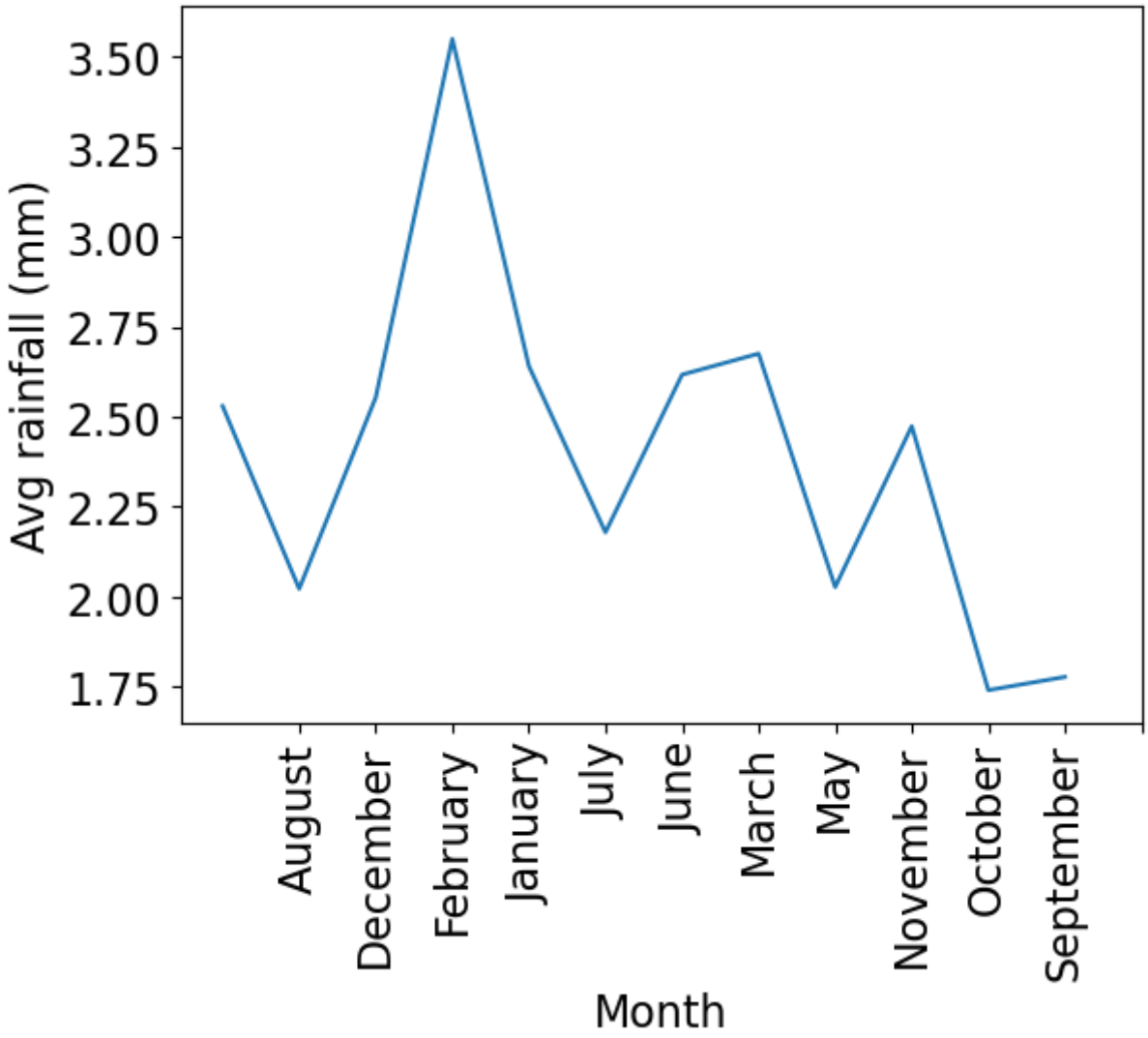
	Coef
x5_Fall	0.068050
x5_Summer	-0.221049
x5_Winter	0.109448
x5_Autumn	0.049084

- No improvements in the scores but the coefficients make some sense,
- A negative coefficient for summer and a positive coefficients for winter.

In [105]:
```
train_df.plot(x="Date", y="Rainfall")
plt.ylabel("Rainfall (mm)");
```



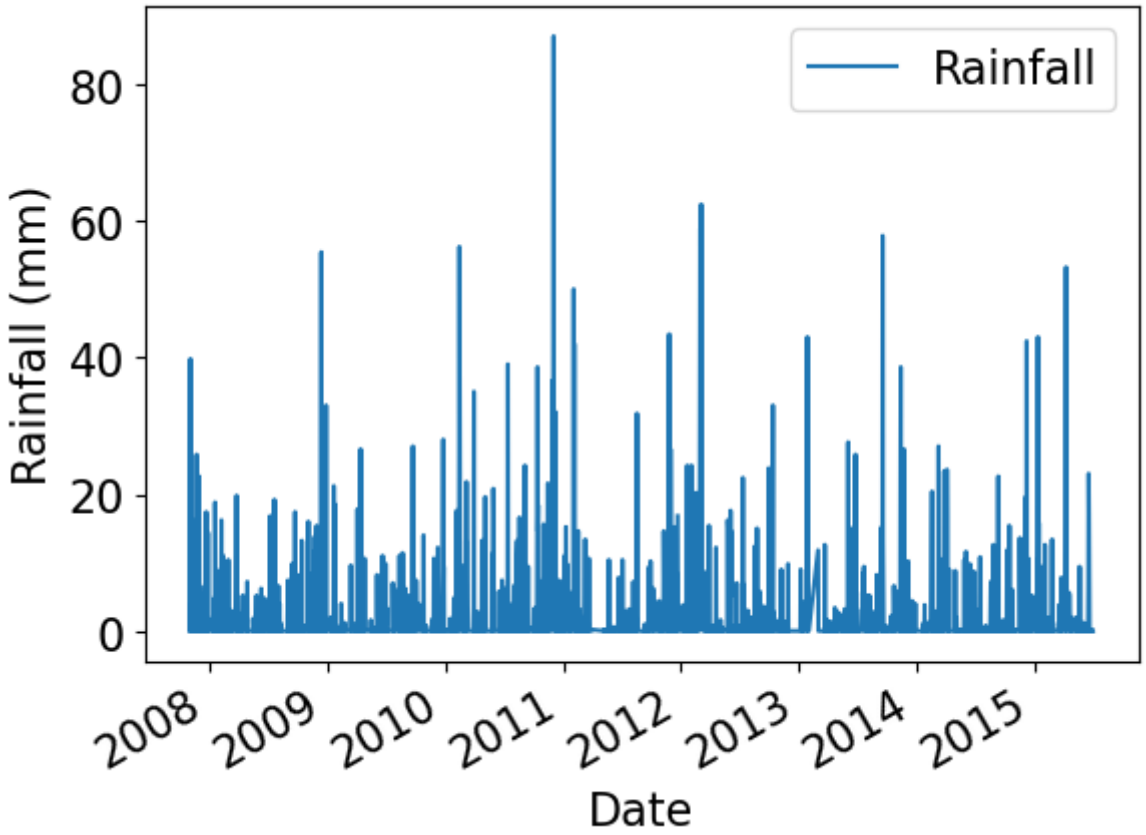
In [106]:
```
monthly_avg_rainfall = train_df.groupby("Month")["Rainfall"].mean()
plt.plot(monthly_avg_rainfall)
plt.xticks(np.arange(1, 13).astype(int))
plt.ylabel("Avg rainfall (mm)")
plt.xlabel("Month")
plt.xticks(rotation=90);
```



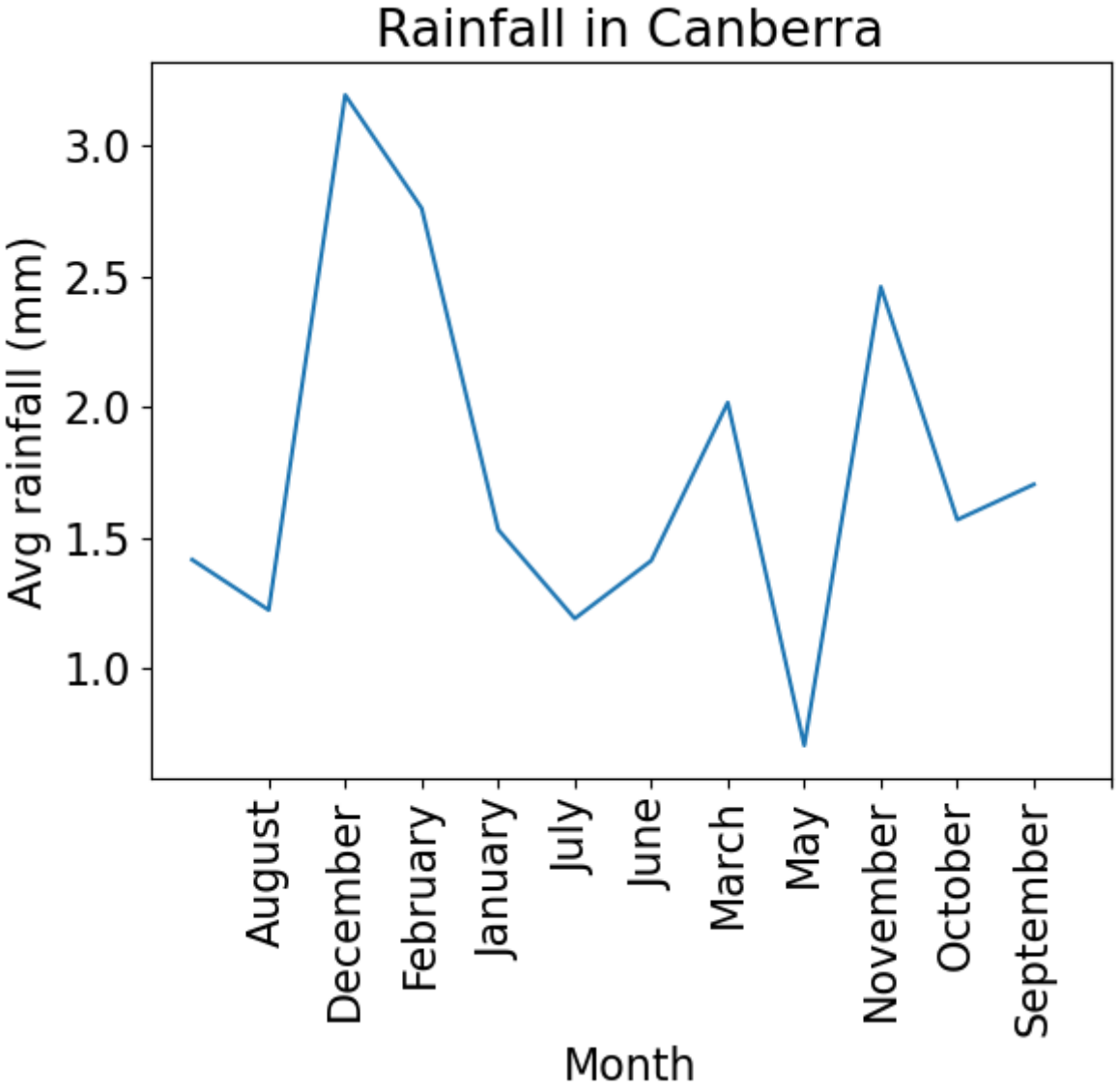
- It's interesting that June rainy but May and August are less so.
- But, Australia is a huge country. Perhaps we should drill down to particular locations:

In [107]:
```
train_df_canberra = train_df.query('Location == "Canberra"')
```

In [108]:
```
train_df_canberra.plot(x="Date", y="Rainfall")
plt.ylabel("Rainfall (mm)");
```



In [109]:
```
plt.plot(train_df_canberra.groupby("Month")["Rainfall"].mean())
plt.xticks(np.arange(1, 13).astype(int))
plt.ylabel("Avg rainfall (mm)")
plt.xlabel("Month")
plt.title("Rainfall in Canberra")
plt.xticks(rotation=90);
```



In [110]:
```
train_df_canberra.shape
```

Out[110]:
```
(2696, 25)
```

- This looks somewhat cleaner but also pretty surprising - why is December so much higher than January?

Lag-based features

- In time series data there is temporal dependence; observations close in time tend to be correlated.

- Currently we're using features about today to predict tomorrow's rainfall.
- But, what if tomorrow's rainfall is also related to yesterday's features, or the day before?
  - This is called a *lagged* feature.
- In time series analysis, we'd look at something called an [autocorrelation function (https://en.wikipedia.org/wiki/Autocorrelation)](https://en.wikipedia.org/wiki/Autocorrelation) (ACF), but we won't go into that here.
- Instead, we can just add those features:

In [111]:
```
train_df = rain_df.query("Date <= 20150630")
test_df = rain_df.query("Date >  20150630")
```

In [112]:
```
train_df
```

Out[112]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGu
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	
...	...	...	...	...	...	...	...	...	
144729	2015-06-26	Uluru	3.8	18.3	0.0	NaN	NaN	E	
144730	2015-06-27	Uluru	2.5	17.1	0.0	NaN	NaN	E	
144731	2015-06-28	Uluru	4.5	19.6	0.0	NaN	NaN	ENE	
144732	2015-06-29	Uluru	7.6	22.0	0.0	NaN	NaN	ESE	
144733	2015-06-30	Uluru	6.8	21.1	0.0	NaN	NaN	ESE	

107502 rows × 23 columns

- It looks like the dataframe is already sorted by Location and then by date for each Location.
- We could have done this ourselves with:

In [113]:
```
# train_df.sort_values(by=["Location", "Date"])
```

But make sure to also sort the targets (i.e. do this before preprocessing).

We can "lag" (or "shift") a time series in Pandas with the .shift() method.

In [114]:
```
train_df = train_df.assign(Rainfall_lag1=train_df["Rainfall"].shift(1))
```

In [115]:
```
train_df[["Date", "Location", "Rainfall", "Rainfall_lag1"]][:20]
```

Out[115]:

	Date	Location	Rainfall	Rainfall_lag1
0	2008-12-01	Albury	0.6	NaN
1	2008-12-02	Albury	0.0	0.6
2	2008-12-03	Albury	0.0	0.0
3	2008-12-04	Albury	0.0	0.0
4	2008-12-05	Albury	1.0	0.0
5	2008-12-06	Albury	0.2	1.0
6	2008-12-07	Albury	0.0	0.2
7	2008-12-08	Albury	0.0	0.0
8	2008-12-09	Albury	0.0	0.0
9	2008-12-10	Albury	1.4	0.0
10	2008-12-11	Albury	0.0	1.4
11	2008-12-12	Albury	2.2	0.0
12	2008-12-13	Albury	15.6	2.2
13	2008-12-14	Albury	3.6	15.6
15	2008-12-16	Albury	NaN	3.6
16	2008-12-17	Albury	0.0	NaN
17	2008-12-18	Albury	16.8	0.0
18	2008-12-19	Albury	10.6	16.8
19	2008-12-20	Albury	0.0	10.6
20	2008-12-21	Albury	0.0	0.0

- But we have multiple time series here and we need to be more careful with this.
- When we switch from one location to another we do not want to take the value from the previous location.

In [116]:
```
def create_lag_feature(df, orig_feature, lag):
    """Creates a new df with a new feature that's a lagged version of t
    # note: pandas .shift() kind of does this for you already, but oh w

    new_df = df.copy()
    new_feature_name = "%s_lag%d" % (orig_feature, lag)
    new_df[new_feature_name] = np.nan
    for location, df_location in new_df.groupby(
        "Location"
    ):  # Each location is its own time series
        new_df.loc[df_location.index[lag:], new_feature_name] = df_loca
            orig_feature
        ].values
    return new_df
```

In [117]:
```
train_df = create_lag_feature(train_df, "Rainfall", 1)
```

In [118]:
```
train_df[["Date", "Location", "Rainfall", "Rainfall_lag1"]][2285:2295]
```

Out[118]:

	Date	Location	Rainfall	Rainfall_lag1
2309	2015-06-26	Albury	0.2	1.0
2310	2015-06-27	Albury	0.0	0.2
2311	2015-06-28	Albury	0.2	0.0
2312	2015-06-29	Albury	0.0	0.2
2313	2015-06-30	Albury	0.0	0.0
3040	2009-01-01	BadgerysCreek	0.0	NaN
3041	2009-01-02	BadgerysCreek	0.0	0.0
3042	2009-01-03	BadgerysCreek	0.0	0.0
3043	2009-01-04	BadgerysCreek	0.0	0.0
3044	2009-01-05	BadgerysCreek	0.0	0.0

Now it looks good!

Question: is it OK to do this to the test set? Discuss.

- It's fine if you would have this information available in deployment.
- If we're just forecasting the next day, we should.
- Let's include it for now.

In [119]:
```
rain_df_modified = create_lag_feature(rain_df, "Rainfall", 1)
train_df = rain_df_modified.query("Date <= 20150630")
test_df = rain_df_modified.query("Date >  20150630")
```

In [120]:
```
rain_df_modified
```

Out[120]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGu
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	
...	...	...	...	...	...	...	...	...	
145454	2017-06-20	Uluru	3.5	21.8	0.0	NaN	NaN	E	
145455	2017-06-21	Uluru	2.8	23.4	0.0	NaN	NaN	E	
145456	2017-06-22	Uluru	3.6	25.3	0.0	NaN	NaN	NNW	
145457	2017-06-23	Uluru	5.4	26.9	0.0	NaN	NaN	N	
145458	2017-06-24	Uluru	7.8	27.0	0.0	NaN	NaN	SE	

142193 rows × 24 columns

In [121]:
```
X_train_enc, y_train, X_test_enc, y_test, preprocessor = preprocess_fea
    train_df,
    test_df,
    numeric_features + ["Rainfall_lag1"],
    categorical_features,
    drop_features,
)
```

In [122]:
```
lr_coef = score_lr_print_coeff(
    preprocessor, train_df, y_train, test_df, y_test, X_train_enc
)
```

```
Train score: 0.85
Test score: 0.84
```

In [123]:
```
lr_coef.loc[["Rainfall", "Rainfall_lag1"]]
```

Out[123]:

	Coef
Rainfall	0.081052
Rainfall_lag1	0.008290

- Rainfall from today has a positive coefficient.

- Rainfall from yesterday has a positive but a smaller coefficient.
- If we didn't have rainfall from today feature, rainfall from yesterday feature would have received a bigger coefficient.

- We could also create a lagged version of the target.
- In fact, this dataset already has that built in! `RainToday` is the lagged version of the target `RainTomorrow`.
- We could also create lagged version of other features, or more lags

In [124]:
```
rain_df_modified = create_lag_feature(rain_df, "Rainfall", 1)
rain_df_modified = create_lag_feature(rain_df_modified, "Rainfall", 2)
rain_df_modified = create_lag_feature(rain_df_modified, "Rainfall", 3)
rain_df_modified = create_lag_feature(rain_df_modified, "Humidity3pm",
```

In [125]:
```
rain_df_modified[
    [
        "Date",
        "Location",
        "Rainfall",
        "Rainfall_lag1",
        "Rainfall_lag2",
        "Rainfall_lag3",
        "Humidity3pm",
        "Humidity3pm_lag1",
    ]
].head(10)
```

Out[125]:

	Date	Location	Rainfall	Rainfall_lag1	Rainfall_lag2	Rainfall_lag3	Humidity3pm	Humidity3pm_lag
0	2008-12-01	Albury	0.6	NaN	NaN	NaN	22.0	Na
1	2008-12-02	Albury	0.0	0.6	NaN	NaN	25.0	22
2	2008-12-03	Albury	0.0	0.0	0.6	NaN	30.0	25
3	2008-12-04	Albury	0.0	0.0	0.0	0.6	16.0	30
4	2008-12-05	Albury	1.0	0.0	0.0	0.0	33.0	16
5	2008-12-06	Albury	0.2	1.0	0.0	0.0	23.0	33
6	2008-12-07	Albury	0.0	0.2	1.0	0.0	19.0	23
7	2008-12-08	Albury	0.0	0.0	0.2	1.0	19.0	19
8	2008-12-09	Albury	0.0	0.0	0.0	0.2	9.0	19
9	2008-12-10	Albury	1.4	0.0	0.0	0.0	27.0	9

Note the pattern of `NaN` values.

In [126]:
```
train_df = rain_df_modified.query("Date <= 20150630")
test_df = rain_df_modified.query("Date >  20150630")
```

In [127]:
```
X_train_enc, y_train, X_test_enc, y_test, preprocessor = preprocess_fea
    train_df,
    test_df,
    numeric_features
    + ["Rainfall_lag1", "Rainfall_lag2", "Rainfall_lag3", "Humidity3pm_
    categorical_features,
    drop_features,
)
```

In [128]:
```
lr_coef = score_lr_print_coeff(
    preprocessor, train_df, y_train, test_df, y_test, X_train_enc
)
```

```
Train score: 0.85
Test score: 0.85
```

In [129]:
```
lr_coef.loc[
    [
        "Rainfall",
        "Rainfall_lag1",
        "Rainfall_lag2",
        "Rainfall_lag3",
        "Humidity3pm",
        "Humidity3pm_lag1",
    ]
]
```

Out[129]:

	Coef
Rainfall	0.108519
Rainfall_lag1	0.023086
Rainfall_lag2	0.018295
Rainfall_lag3	0.017829
Humidity3pm	1.278602
Humidity3pm_lag1	-0.267480

Note the pattern in the magnitude of the coefficients.

In [ ]:
```

```

Forecasting further into the future

- Let's say we want to predict 7 days into the future instead of one day.
- There are a few main approaches here:

1. Train a separate model for each number of days. E.g. one model that predicts RainTomorrow, another model that predicts RainIn2Days, etc. We can build these datasets.
2. Use a multi-output model that jointly predicts RainTomorrow, RainIn2Days, etc. However, multi-output models are outside the scope of CPSC 330.
3. Use one model and sequentially predict using a `for` loop. However, this requires predicting *all* features into a model so may not be that useful here.

- To briefly dig into approach 3, this is easier to understand for a univariate (one feature) time series.
- To dig into this we'll look at the Retail Sales of Clothing and Clothing Accessory Stores dataset (https://fred.stlouisfed.org/series/MRTSSM448USN) made available by the Federal Reserve Bank of St. Louis.

In [130]:
```
retail_df = pd.read_csv("../data/MRTSSM448USN.csv", parse_dates=["DATE"
retail_df.columns = ["date", "sales"]
```

In [131]:
```
retail_df.head()
```

Out[131]:

	date	sales
0	1992-01-01	6938
1	1992-02-01	7524
2	1992-03-01	8475
3	1992-04-01	9401
4	1992-05-01	9558

In [132]:
```
retail_df["date"].min()
```

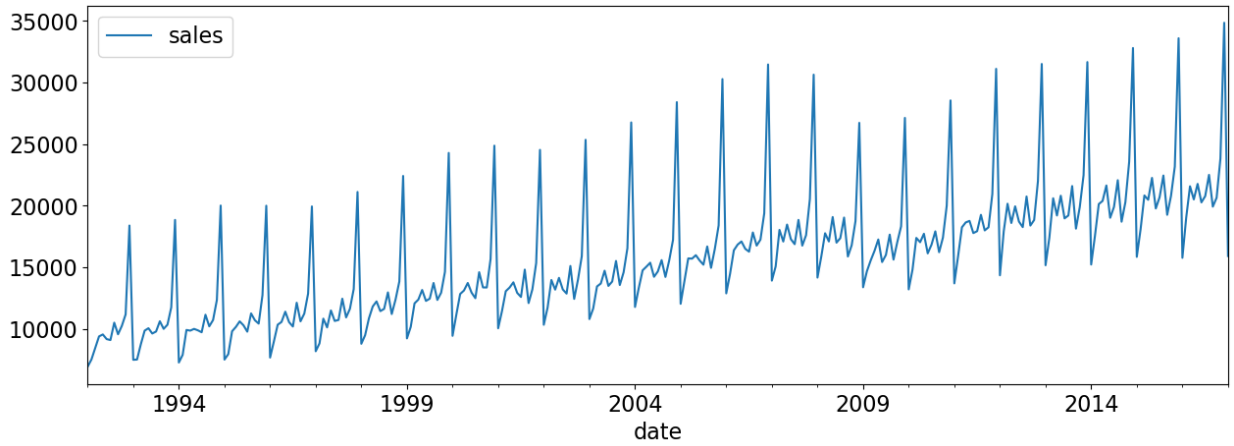
Out[132]:
```
Timestamp('1992-01-01 00:00:00')
```

In [133]:
```
retail_df["date"].max()
```

Out[133]:
```
Timestamp('2023-01-01 00:00:00')
```

In [134]:
```
retail_df_train = retail_df.query("date <= 20170101")
retail_df_test = retail_df.query("date >  20170101")
```

In [135]:
```
retail_df_train.plot(x="date", y="sales", figsize=(15, 5));
```



We can create a dataset using purely lag features.

In [136]:
```
def lag_df(df, lag, cols):
    return df.assign(
        **{f"{col}-{n}": df[col].shift(n) for n in range(1, lag + 1) fo
    )
```

In [137]:
```
retail_lag_5 = lag_df(retail_df, 5, ["sales"])
retail_train_5 = retail_lag_5.query("date <= 20170101")
retail_test_5 = retail_lag_5.query("date >  20170101")
retail_train_5
```

Out[137]:

	date	sales	sales-1	sales-2	sales-3	sales-4	sales-5
0	1992-01-01	6938	NaN	NaN	NaN	NaN	NaN
1	1992-02-01	7524	6938.0	NaN	NaN	NaN	NaN
2	1992-03-01	8475	7524.0	6938.0	NaN	NaN	NaN
3	1992-04-01	9401	8475.0	7524.0	6938.0	NaN	NaN
4	1992-05-01	9558	9401.0	8475.0	7524.0	6938.0	NaN
...	...	...	...	...	...	...	...
296	2016-09-01	19928	22505.0	20782.0	20274.0	21774.0	20514.0
297	2016-10-01	20650	19928.0	22505.0	20782.0	20274.0	21774.0
298	2016-11-01	23826	20650.0	19928.0	22505.0	20782.0	20274.0
299	2016-12-01	34847	23826.0	20650.0	19928.0	22505.0	20782.0
300	2017-01-01	15921	34847.0	23826.0	20650.0	19928.0	22505.0

301 rows × 7 columns

- Now, if we drop the "date" column we have a target ("sales") and 5 features (the previous 5 days of sales).
- We need to impute/drop the missing values and then we can fit a model to this. I will just drop for convenience:

In [138]:
```
retail_train_5 = retail_train_5[5:].drop(columns=["date"])
retail_train_5
```

Out[138]:

	sales	sales-1	sales-2	sales-3	sales-4	sales-5
5	9182	9558.0	9401.0	8475.0	7524.0	6938.0
6	9103	9182.0	9558.0	9401.0	8475.0	7524.0
7	10513	9103.0	9182.0	9558.0	9401.0	8475.0
8	9573	10513.0	9103.0	9182.0	9558.0	9401.0
9	10254	9573.0	10513.0	9103.0	9182.0	9558.0
...	...	...	...	...	...	...
296	19928	22505.0	20782.0	20274.0	21774.0	20514.0
297	20650	19928.0	22505.0	20782.0	20274.0	21774.0
298	23826	20650.0	19928.0	22505.0	20782.0	20274.0
299	34847	23826.0	20650.0	19928.0	22505.0	20782.0
300	15921	34847.0	23826.0	20650.0	19928.0	22505.0

296 rows × 6 columns

In [139]:
```
retail_train_5_X = retail_train_5.drop(columns=["sales"])
retail_train_5_y = retail_train_5["sales"]
```

In [140]:
```
from sklearn.ensemble import RandomForestRegressor
```

In [141]:
```
retail_model = RandomForestRegressor()
retail_model.fit(retail_train_5_X, retail_train_5_y);
```

Given this, we can now predict the sales

In [142]:
```
preds = retail_model.predict(retail_test_5.drop(columns=["date", "sales
preds
```

Out[142]:
```
array([18768.09, 20491.71, 20789.32, 22529.24, 20617.  , 21838.07,
       22218.1 , 22352.96, 20851.81, 22713.  , 31629.71, 15989.66,
       18587.88, 21489.34, 22392.5 , 21835.42, 28448.45, 20629.66,
       22276.94, 29463.9 , 21449.94, 22054.79, 21093.12, 15949.61,
       18826.44, 20505.58, 22669.61, 22053.39, 27977.7 , 22124.38,
       22006.33, 30543.09, 21478.95, 21951.6 , 28344.54, 15952.53,
       19153.75, 22005.46, 13647.61, 11574.87, 11231.21, 15020.43,
       12481.92, 11072.85, 18325.01, 20567.68, 22414.79, 14965.93,
       17696.88, 18930.01, 31319.99, 29013.66, 15351.69, 29359.58,
       26889.1 , 28723.33, 22943.54, 30084.94, 16856.13, 17222.57,
       20365.81, 22563.73, 19718.08, 17245.92, 17350.4 , 29038.05,
       26889.1 , 18079.17, 30265.52, 28723.33, 16856.13, 17222.57])
```

In [143]:
```
retail_test_5_preds = retail_test_5.assign(predicted_sales=preds)
retail_test_5_preds.head()
```

Out[143]:

	date	sales	sales-1	sales-2	sales-3	sales-4	sales-5	predicted_sales
301	2017-02-01	18036	15921.0	34847.0	23826.0	20650.0	19928.0	18768.09
302	2017-03-01	21348	18036.0	15921.0	34847.0	23826.0	20650.0	20491.71
303	2017-04-01	21154	21348.0	18036.0	15921.0	34847.0	23826.0	20789.32
304	2017-05-01	21954	21154.0	21348.0	18036.0	15921.0	34847.0	22529.24
305	2017-06-01	20623	21954.0	21154.0	21348.0	18036.0	15921.0	20617.00

- Ok, that is fine, but what if we want to predict 7 days in the future?
- Well, we would not have access to our features!! We don't yet know the previous day's sales, or 2 days prior!
- So we can use "Approach 3" mentioned earlier: predict these values and then pretend they are true!
- For simplicity, say today is Monday

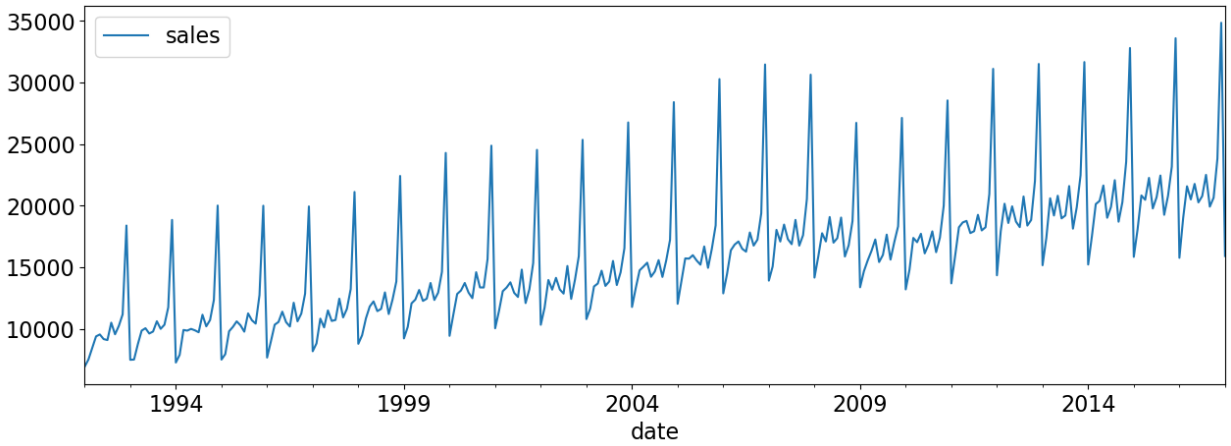
1. Predict Tuesday's sales
2. Then, to predict for Wednesday, we need to know Tuesday's sales. Use our *prediction* for Tuesday as the truth.
3. Then, to predict for Thursday, we need to know Tue and Wed sales. Use our predictions.
4. Etc etc.

In [ ]:
```

```

Trends

- There are some important concepts in time series that rely on having a continuous target (like we do in the retail sales example above).
- Part of that is the idea of seasonality and trends.
- These are mostly taken care of by our feature engineering of the data variable, but there's something important left to discuss.

In [144]:
```
1 retail_df_train.plot(x="date", y="sales", figsize=(15, 5));
```



- It looks like there's a **trend** here - the sales are going up over time.

Let's say we encoded the date as a feature in days like this:

In [145]:
```
1 retail_train_5_date = retail_lag_5.query("date <= 20170101")
2 first_day_retail = retail_train_5_date["date"].min()
3
4 retail_train_5_date = retail_train_5_date.assign(
5     Days_since=retail_train_5_date["date"].apply(lambda x: (x - first_d
6 )
7 retail_train_5_date.head(10)
```

Out[145]:

	date	sales	sales-1	sales-2	sales-3	sales-4	sales-5	Days_since
0	1992-01-01	6938	NaN	NaN	NaN	NaN	NaN	0
1	1992-02-01	7524	6938.0	NaN	NaN	NaN	NaN	31
2	1992-03-01	8475	7524.0	6938.0	NaN	NaN	NaN	60
3	1992-04-01	9401	8475.0	7524.0	6938.0	NaN	NaN	91
4	1992-05-01	9558	9401.0	8475.0	7524.0	6938.0	NaN	121
5	1992-06-01	9182	9558.0	9401.0	8475.0	7524.0	6938.0	152
6	1992-07-01	9103	9182.0	9558.0	9401.0	8475.0	7524.0	182
7	1992-08-01	10513	9103.0	9182.0	9558.0	9401.0	8475.0	213
8	1992-09-01	9573	10513.0	9103.0	9182.0	9558.0	9401.0	244
9	1992-10-01	10254	9573.0	10513.0	9103.0	9182.0	9558.0	274

- Now, let's say we use all these features (the lagged version of the target and also `Days_since`.
- If we use **linear regression** we'll learn a coefficient for `Days_since`.
  - If that coefficient is positive, it predicts unlimited growth forever. That may not be what you want? It depends.

- If we use a **random forest**, we'll just be doing splits from the training set, e.g. "if `Days_since` > 9100 then do this".
  - There will be no splits for later time points because there is no training data there.
  - Thus tree-based models cannot model trends.
  - This is really important to know!!
- Often, we model the trend separately and use the random forest to model a de-trended time series.

In [ ]:
```
1
```

What did we not cover? (5 min)

- A huge amount!

Traditional time series approaches

- Time series analysis is a huge field of its own (notice a pattern here?)
- Traditional approaches include the ARIMA model (https://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average) and its various components/extensions.
- In Python, the statsmodels (https://www.statsmodels.org/) package is the place to go for this sort of thing.
  - For example, statsmodels.tsa.arima_model.ARIMA (https://www.statsmodels.org/stable/generated/statsmodels.tsa.arima_model.ARIMA.html).
- These approaches can forecast, but they are also very good for understanding the temporal relationships in your data.
- We will take a different route in this course, and stick to our supervised learning tools.

Deep learning

- Recently, deep learning has been very successful too.
- In particular, recurrent neural networks (https://en.wikipedia.org/wiki/Recurrent_neural_network) (RNNs).
  - These are not covered in CPSC 340, but I believe they are in 440.
  - LSTMs (https://en.wikipedia.org/wiki/Long_short-term_memory) especially have shown a lot of promise in this type of task.
  - Here (https://colah.github.io/posts/2015-08-Understanding-LSTMs/) is a blog post about LSTMs.

Types of problems involving time series

- A single label associated with an entire time series.
  - We had that with images earlier on, you could have the same for a time series.
  - E.g., for fraud detection, labelling each transaction as fraud/normal vs. labelling a person as bad/good based on their entire history.

    - There are various approaches that can be used for this type of problem, including CNNs, LSTMs, and non deep learning methods.
- Inference problems.
    - What are the patterns in this time series?
    - How many lags are associated with the current value?
- Etc.

Unequally spaced time points

- We assumed we have a measurement each day.
- For example, when creating lag features we used consecutive rows in the DataFrame.
- But, in fact some days were missing in this dataset.
- More generally, what if the measurements are at arbitrary times, not equally spaced?
    - Some of our approaches would still work, like encoding the month / looking at seasonality.
    - Some of our approaches would not make sense, like the lags.
    - Perhaps the measurements could be binned into equally spaced bins, or something.
    - This is more of a hassle.

Other software package

- One good one to know about is Prophet (https://facebook.github.io/prophet/docs/quick_start.html).

Feature engineering

- Often, a useful approach is to just *engineer your own features*.
    - E.g., max expenditure, min expenditure, max-min, avg time gap between transactions, variance of time gap between transactions, etc etc.
    - We could do that here as well, or in any problem.